Workload Design: Selecting Representative Program-Input Pairs

Lieven Eeckhout Hans Vandierendonck Koen De Bosschere
Department of Electronics and Information Systems (ELIS), Ghent University, Belgium
E-mail: {leeckhou,hvdieren,kdb}@elis.rug.ac.be

Abstract

Having a representative workload of the target domain of a microprocessor is extremely important throughout its design. The composition of a workload involves two issues: (i) which benchmarks to select and (ii) which input data sets to select per benchmark. Unfortunately, it is impossible to select a huge number of benchmarks and respective input sets due to the large instruction counts per benchmark and due to limitations on the available simulation time. In this paper, we use statistical data analysis techniques such as principal components analysis (PCA) and cluster analysis to efficiently explore the workload space. Within this workload space, different input data sets for a given benchmark can be displayed, a distance can be measured between program-input pairs that gives us an idea about their mutual behavioral differences and representative input data sets can be selected for the given benchmark. This methodology is validated by showing that program-input pairs that are close to each other in this workload space indeed exhibit similar behavior. The final goal is to select a limited set of representative benchmark-input pairs that span the complete workload space. Next to workload composition, there are a number of other possible applications, namely getting insight in the impact of input data sets on program behavior and profile-guided compiler optimizations.

1 Introduction

The first step when designing a new microprocessor is to compose a workload that should be representative for the set of applications that will be run on the microprocessor once it will be used in a commercial product [1, 7]. A workload then typically consists of a number of benchmarks with respective input data sets taken from various benchmarks suites, such as SPEC, TPC, MediaBench, etc. This workload will then be used during the various simulation runs to perform design space explorations. It is obvious that *workload design*, or composing a representative workload, is extremely important in order to obtain a microprocessor design that is optimal for the target environment of operation. The question when composing a representative workload is thus twofold: (i) which benchmarks and (ii) which input data sets to select. In addition, we have to take into account that even high-level architectural simulations are extremely time-consuming. As such, the total simulation time should be limited as much as possible to limit the time-to-market. This implies that the total number of benchmarks and input data sets should be limited without compromising the final design. Ideally, we would like to have a limited set of benchmark-input pairs spanning the complete workload space, which contains a variety of the most important types of program behavior.

Conceptually, the complete workload design space can be viewed as a p-dimensional space with p the number of important program characteristics that affect performance, e.g., branch prediction accuracy, cache miss rates, instruction-level parallelism, etc. Obviously, p will be too large to display the workload design space understandably. In addition, correlation exists between these variables which reduces the ability to understand what program characteristics are fundamental to make the diversity in the workload space. In this paper, we reduce the p-dimensional workload space to a q-dimensional space with $q \ll p$ ($q = 2$ to $q = 4$ typically) making the visualisation of the workload space possible without losing important information. This is achieved by using statistical data reduction techniques such as principal components analysis (PCA) and cluster analysis.

Each benchmark-input pair is a point in this (reduced) q-dimensional space obtained after PCA. We can expect that different benchmarks will be 'far away' from each other while different input data sets for a single benchmark will be clustered together. This representation gives us an excellent opportunity to measure the impact of input data sets on program behavior. Weak clustering (for various inputs and a single benchmark) indicates that the input set has a large impact on program behavior; strong clustering on the other hand, indicates a small impact. This claim is validated by showing that program-input pairs that are close to each other in the workload space indeed exhibit similar behavior. I.e., 'close' program-input pairs react in similar ways when changes are made to the architecture.

In addition, this representation gives us an idea which input sets should be selected when composing a workload. Strong clustering suggests that a single or only a few input sets could be selected to be representative for the cluster. This will reduce the total simulation time significantly for two reasons: (i) the total number of benchmark-input pairs is reduced; and (ii) we can select the benchmark-input pair with the smallest dynamic instruction count among all the

pairs in the cluster. The reduction of the total simulation time is an important issue for the evaluation of microprocessor designs since todays and future workloads tend to have large dynamic instruction counts.

Another important application, next to getting insight in program behavior and workload composition, is profile-driven compiler optimizations. During profile-guided optimizations, the compiler uses information from previous program runs (obtained through profiling) to guide compiler optimizations. Obviously, for effective optimizations, the input set used for obtaining this profiling information should be representative for a large set of possible input sets. The methodology proposed in this paper can be useful in this respect because input sets that are close to each other in the workload space will have similar behavior.

This paper is organized as follows. In section 2, the program characteristics used are enumerated. Principal components analysis, cluster analysis and their use in this context are discussed in section 3. In section 4, it is shown that these data reduction techniques are useful in the context of workload characterization. In addition, we discuss how input data sets affect program behavior. Section 5 discusses related work. We conclude in section 6.

2 Workload Characterization

It is important to select program characteristics that affect performance for performing data analysis techniques in the context of workload characterization. Selecting program characteristics that do not affect performance, such as the dynamic instruction count, might discriminate benchmark-input pairs on a characteristic that does not affect performance, yielding no information about the behavior of the benchmark-input pair when executed on a microprocessor. On the other hand, it is important to incorporate as many program characteristics as possible so that the analysis done on it will be predictive. I.e., we want strongly clustered program-input pairs to behave similarly so that a single program-input pair can be chosen as a representative of the cluster. The determination of what program characteristics to be included in the analysis in order to obtain a predictive analysis is a study on its own and is out of the scope of this paper. The goal of this paper is to show that data analysis techniques such as PCA and cluster analysis can be a helpful tool for getting insight in the workload space when composing a representative workload.

We have identified the following program characteristics:

- **Instruction mix.** We consider five instruction classes: integer arithmetic operations, logical operations, shift and byte manipulation operations, load/store operations and control operations.
- **Branch prediction accuracy.** We consider the branch prediction accuracy of three branch predictors: a bimodal branch predictor, a gshare branch predictor and a hybrid branch predictor. The bimodal branch predictor consists of an 8K-entry table containing 2-bit saturating counters which is indexed by the program counter of the branch. The gshare branch predictor is an 8K-entry table with 2-bit saturating counters indexed by the program counter xor-ed with the taken/not-taken branch history of 12 past branches. The hybrid branch predictor [10] combines the bimodal and the gshare predictor by choosing among them dynamically. This is done using a meta predictor that is indexed by the branch address and contains 8K 2-bit saturating counters.
- **Data cache miss rates.** Data cache miss rates were measured for five different cache configurations: an 8KB and a 16KB direct mapped cache, a 32KB and a 64KB two-way set-associative cache and a 128KB four-way set-associative cache. The block size was set to 32 bytes.
- **Instruction cache miss rates.** Instruction cache miss rates were measured for the same cache configurations mentioned for the data cache.
- **Sequential flow breaks.** We have also measured the number of instructions between two sequential flow breaks or, in other words, the number of instructions between two taken branches. Note that this metric is higher than the basic block size because some basic blocks 'fall through' to the next basic block.
- **Instruction-level parallelism.** To measure the amount of ILP in a program, we consider an infinite-resource machine, i.e., infinite number of functional units, perfect caches, perfect branch prediction, etc. In addition, we schedule instructions as soon as possible assuming unit execution instruction latency. The only dependencies considered between instructions are read-after-write (RAW) dependencies through registers as well as through memory. In other words, perfect register and memory renaming are assumed in these measurements.

For this study, there are $p = 20$ program characteristics in total on which the analyses are done.

3 Data Analysis

In the first two subsections of this section, we will discuss two data analysis techniques, namely principal components analysis (PCA) and cluster analysis. In the last subsection, we will detail how we used these techniques for analyzing the workload space in this study.

3.1 Principal Components Analysis

Principal components analysis (PCA) [9] is a statistical data analysis technique that presents a different view on the measured data. It builds on the assumption that many variables (in our case, program characteristics) are correlated and hence, they measure the same or similar properties of the program-input pairs. PCA computes new

variables, called *principal components*, which are *linear combinations* of the original variables, such that all principal components are uncorrelated. PCA tranforms the p variables $X_1, X_2, \ldots, X_p$ into p principal components $Z_1, Z_2, \ldots, Z_p$ with $Z_i = \sum_{j=1}^{p} a_{ij} X_j$. This transformation has the properties (i) $Var[Z_1] > Var[Z_2] > \ldots > Var[Z_p]$ which means that Z_1 contains the most information and Z_p the least; and (ii) $Cov[Z_i, Z_j] = 0, \forall i \neq j$ which means that there is no information overlap between the principal components. Note that the total variance in the data remains the same before and after the transformation, namely $\sum_{i=1}^{p} Var[X_i] = \sum_{i=1}^{p} Var[Z_i]$.

As stated in the first property in the previous paragraph, some of the principal components will have a high variance while others will have a small variance. By removing the components with the lowest variance from the analysis, we can reduce the number of program characteristics while controlling the amount of information that is thrown away. We retain q principal components which is a significant information reduction since $q \ll p$ in most cases, typically $q = 2$ to $q = 4$. To measure the fraction of information retained in this q-dimensional space, we use the amount of variance $(\sum_{i=1}^{q} Var[Z_i])/(\sum_{i=1}^{p} Var[X_i])$ accounted for by these q principal components.

In this study the p original variables are the program characteristics mentioned in section 2. By examining the most important q principal components, which are linear combinations of the original program characteristics, meaningful interpretations can be given to these principal components in terms of the original program characteristics. To facilitate the interpretation of the principal components, we apply the *varimax* rotation [9]. This rotation makes the coefficients a_{ij} either close to ± 1 or zero, such that the original variables either have a strong impact on a principal component or they have no impact. Note that varimax rotation is an orthogonal transformation which implies that the rotated principal components are still uncorrelated.

The next step in the analysis is to display the various benchmarks as points in the q-dimensional space built up by the q principal components. This can be done by computing the values of the q retained principal components for each program-input pair. As such, a view can be given on the workload design space and the impact of input data sets on program behavior can be displayed, as will be discussed in the evaluation section of this paper.

During principal components analysis, one can either work with normalized or non-normalized data (the data is normalized when the mean of each variable is zero and its variance is one). In the case of non-normalized data, a higher weight is given in the analysis to variables with a higher variance. In our experiments, we have used normalized data because of our heterogeneous data; e.g., the variance of the ILP is orders of magnitude larger than the variance of the data cache miss rates.

3.2 Cluster Analysis

Cluster analysis [9] is another data analysis technique that is aimed at clustering n cases, in our case program-input pairs, based on the measurements of p variables, in our case program characteristics. The final goal is to obtain a number of groups, containing program-input pairs that have 'similar' behavior. A commonly used algorithm for doing cluster analysis is *hierarchic clustering* which starts with a matrix of distances between the n cases or program-input pairs. As a starting point for the algorithm, each program-input pair is considered as a group. In each iteration of the algorithm, the two groups that are most close to each other (with the smallest distance, also called the *linkage distance*) will be combined to form a new group. As such, close groups are gradually merged until finally all cases will be in a single group. This can be represented in a so called *dendrogram*, which graphically represents the linkage distance for each group merge at each iteration of the algorithm. Having obtained a dendrogram, it is up to the user to decide how many clusters to take. This decision can be made based on the linkage distance. Indeed, small linkage distances imply strong clustering while large linkage distances imply weak clustering.

How we define the distance between two program-input pairs will be explained in the next section. To compute the distance between two groups, we have used the *nearest neighbour* strategy or *single linkage*. This means that the distance between two groups is defined as the smallest distance between two members of each group.

3.3 Workload Analysis

The workload analysis done in this paper is a combination of PCA and cluster analysis and consists of the following steps:

1. The $p = 20$ program characteristics as discussed in section 2 are measured by instrumenting the benchmark programs with ATOM [13], a binary instrumentation tool for the Alpha architecture. With ATOM, a statically linked binary can be transformed to an instrumented binary. Executing this instrumented binary on an Alpha machine yields us the program characteristics that will be used throughout the analysis. Measuring these $p = 20$ program characteritics was done for the 79 program-input pairs mentioned in section 4.1.

2. In a second step, these 79 (number of program-input pairs) $\times 20$ ($= p$, number of program characteristics) data points are normalized so that for each program characteristic the average equals zero and the variance equals one. On these normalized data points, principal components analysis (PCA) is done using STATISTICA [14], a package for statistical computations. This works as follows. A 2-dimensional matrix is presented as input to STATISTICA that has 20 columns representing the original program characteristics as mentioned in section 2. There are 79 rows in this matrix representing the various program-input pairs. On this matrix, PCA is performed by STATISTICA which yields us p principal components.

3. Once these p principal components are obtained, a varimax rotation can be done on these data for improving the understanding of the principal components. This can be done using STATISTICA.

4. Now, it is up to the user to determine how many principal components need to be retained. This decision is made based on the amount of variance accounted for by the retained principal components.

5. The q principal components that are retained can be analyzed and a meaningful interpretation can be given to them. This is done based on the coefficients a_{ij}, also called the *factor loadings*, as they occur in the following equation $Z_i = \sum_{j=1}^{p} a_{ij} X_j$, with $Z_i, 1 \leq i \leq q$ the principal components and $X_j, 1 \leq j \leq p$ the original program characteristics. A positive coefficient a_{ij} means a positive impact of program characteristic X_j on principal component Z_i; a negative coefficient a_{ij} implies a negative impact. If a coefficient a_{ij} is close to zero, this means X_j has (nearly) no impact on Z_i.

6. The program-input pairs can be displayed in the workload space built up by these q principal components. This can easily be done by computing $Z_i = \sum_{j=1}^{p} a_{ij} X_j$ for each program-input pair.

7. Within this q-dimensional space the Euclidean distance can be computed between the various program-input pairs as a reliable measure for the way program-input pairs differ from each other. There are two reasons supporting this statement. First, the values along the axes in this space are uncorrelated since they are determined by the principal components which are uncorrelated by construction. The absence of correlation is important when calculating the Euclidean distance because two correlated variables—that essentially measure the same thing—will contribute a similar amount to the overall distance as an independent variable; as such, these variables would be counted twice, which is undesirable. Second, the variance along the q principal components is meaningful since it is a measure for the diversity along each principal component by construction.

8. Finally, cluster analysis can be done using the distance between program-input pairs as determined in the previous step. Based on the dendrogram a clear view is given on the clustering within the workload space.

The reason why we chose to first perform PCA and subsequently cluster analysis instead of applying cluster analysis on the initial data is as follows. The original variables are highly correlated which implies that an Euclidean distance in this space is unreliable due to this correlation as explained previously. The most obvious solution would have been to use the *Mahalanobis* distance [9] which takes into account the correlation between the variables. However, the computation of the Mahalanobis distance is based on a pooled *estimate* of the common covariance matrix which might introduce inaccuracies.

4 Evaluation

In this section, we first present the program-input pairs that are used in this study. Second, we show the results of performing the workload analysis as discussed in section 3.3. Finally, the methodology is validated in section 4.3.

4.1 Experimental Setup

In this study, we have used the SPECint95 benchmarks (`http://www.spec.org`) and a database workload consisting of TPC-D queries (`http://www.tpc.org`), see Table 1. The reason why we chose SPECint95 instead of the more recent SPECint2000 is to limit the simulation time. SPEC opted to dramatically increase the runtimes of the SPEC2000 benchmarks compared to the SPEC95 benchmarks which is beneficial for performance evaluation on real hardware but impractical for simulation purposes. In addition, there are more reference inputs provided with SPECint95 then with SPECint2000. For gcc (GNU C compiler) and li (lisp interpreter), we have used all the reference input files. For ijpeg (image processing), penguin, specmun and vigo were taken from the reference input set. The other images that served as input to ijpeg were taken from the web. The dimensions of the images are shown in brackets. For compress (text compression), we have adapted the reference input '14000000 e 2231' to obtain different input sets. For m88ksim (microprocessor simulation) and vortex (object-oriented database), we have used the train and the reference inputs. The same was done for perl (perl interpreter): jumble was taken from the train input, and primes and scrabbl were taken from the reference input; as well as for go (game): '50 9 2stone9.in' from the train input, and '50 21 9stone21.in' and '50 21 5 stone21.in' from the reference input.

In addition to SPECint95, we used postgres v6.3 running the decision support TPC-D queries over a 100MB Btree-indexed database. For postgres, we ran all TPC-D queries except for query 1 because of memory constraints on our machine.

The benchmarks were compiled with optimization level -O4 and linked statically with the -non_shared flag for the Alpha architecture.

4.2 Results

In this section, we will first perform PCA on the data for the various input sets of gcc. Subsequently, the same will be done for postgres. Finally, PCA and cluster analysis will be applied on the data for all the benchmark-input pairs of Table 1. We present the data for gcc and postgres before presenting the analysis of all the program-input pairs because these two benchmarks illustrate different aspects of the techniques in terms of the number of principal components, clustering, etc.

benchmark	input	dyn (M)	stat	mem (K)
gcc	amptjp	835	147,402	375
	c-decl-s	835	147,369	375
	cccp	886	145,727	371
	cp-decl	1,103	143,153	579
	dbxout	141	120,057	215
	emit-rtl	104	127,974	108
	explow	225	105,222	280
	expr	768	142,308	653
	gcc	141	129,852	125
	genoutput	74	117,818	104
	genrecog	100	124,362	133
	insn-emit	126	84,777	199
	insn-recog	409	105,434	357
	integrate	188	133,068	199
	jump	133	126,400	130
	print-tree	136	118,051	201
	protoize	298	137,636	159
	recog	227	123,958	161
	regclass	91	125,328	117
	reload1	778	146,076	542
	stmt-protoize	654	148,026	261
	stmt	356	138,910	250
	toplev	168	125,810	218
	varasm	166	139,847	168
postgres	Q2	227	57,297	345
	Q3	948	56,676	358
	Q4	564	53,183	285
	Q5	7,015	60,519	654
	Q6	1,470	46,271	1,080
	Q7	932	69,551	631
	Q8	842	61,425	11,821
	Q9	9,343	68,837	10,429
	Q10	1,794	62,564	681
	Q11	188	65,747	572
	Q12	1,770	65,377	258
	Q13	325	65,322	264
	Q14	1,440	67,966	448
	Q15	1,641	67,246	640
	Q16	82,228	58,067	389
	Q17	183	54,835	366

benchmark	input	dyn (M)	stat	mem (K)
li	boyer	226	9,067	36
	browse	672	9,607	39
	ctak	583	8,106	18
	dderiv	777	9,200	16
	deriv	719	8,826	15
	destru2	2,541	9,182	16
	destrum2	2,555	9,182	16
	div2	2,514	8,546	19
	puzzle0	2	8,728	19
	tak2	6,892	8,079	16
	takr	1,125	8,070	36
	triang	3	9,008	15
ijpeg	band (2362x1570)	2,934	16,183	5,718
	beach (512x480)	254	16,039	405
	building (1181x1449)	1,626	16,224	2,742
	car (739x491)	373	16,294	596
	dessert (491x740)	353	16,267	587
	globe (512x512)	274	16,040	436
	kitty (512x482)	267	16,088	412
	monalisa (459x703)	259	16,160	508
	penguin (1024x739)	790	16,128	1,227
	specmun (1024x688)	730	15,952	1,136
	vigo (1024x768)	817	16,037	1,273
compress	14000000 e 2231 (ref)	60,102	4,507	4,601
	10000000 e 2231	42,936	4,507	3,318
	5000000 e 2231	21,495	4,494	1,715
	1000000 e 2231	4,342	4,490	433
	500000 e 2231	2,182	4,496	272
	100000 e 2231	423	4,361	142
m88ksim	train	24,959	11,306	4,834
	ref	71,161	14,287	4,834
vortex	train	3,244	78,766	1,266
	ref	92,555	78,650	5,117
perl	jumble	2,945	21,343	5,951
	primes	17,375	16,527	8
	scrabbl	28,251	21,674	4,098
go	50 9 2stone9.in	593	55,894	45
	50 21 9stone21.in	35,758	62,435	57
	50 21 5stone21.in	35,329	62,841	57

Table 1. Characteristics of the benchmarks used with their inputs, dynamic instruction count (in millions), static instruction count (number of instructions executed at least once) and data memory footprint in 64-bit words (in thousands).

4.2.1 Gcc

PCA and varimax rotation extract two principal components from the data of gcc with 24 input sets. These two principal components together account for 96.9% of the total variance; the first and the second component account for 49.6% and 47.3% of the total variance, respectively. In Figure 1, the factor loadings are presented for these two principal components. E.g., this means that the first principal component is computed as $PC1 = 0.43 \times ILP + 0.94 \times bimodal + 0.94 \times gshare + \ldots$. The first component is positively dominated, see Figure 1, by the branch prediction accuracy, the percentage of arithmetic and logical operations; and negatively dominated by the I-cache miss rates. The second component is positively dominated by the D-cache miss rates, the percentage of shift and control operations; and negatively dominated by the ILP, the percentage of load/store operations and the number of instructions between two taken branches. Figure 2 presents the various input sets of gcc in the 2-dimensional space built up by these two components. Data points in this graph with a high value along the first component, have high branch prediction accuracies and high percentages of arithmetic and logical operations compared to the other data points; in addition, these data points also have low I-cache miss rates. Note that these data are normalized. Thus, only relative distances are important. For example, emit-rtl and insn-emit are relatively closer to each other than emit-rtl and cp-decl.

Figure 2 shows that gcc executing input set explow exhibits a different behavior than the other input sets. This is due to its high D-cache miss rates, its high percentage of shift and control operations, and its low ILP, its low percentage of load/store operations and its low number of instructions between two taken branches. The input sets emit-rtl and insn-emit have a high I-cache miss rate, a low branch prediction accuracy and a low percentage of arithmetic and logical operations; for reload1 the opposite is true. This can be concluded from the factor loadings presented in Figure 1; we also verified that this is true by inspecting the original data. The strong cluster in the middle

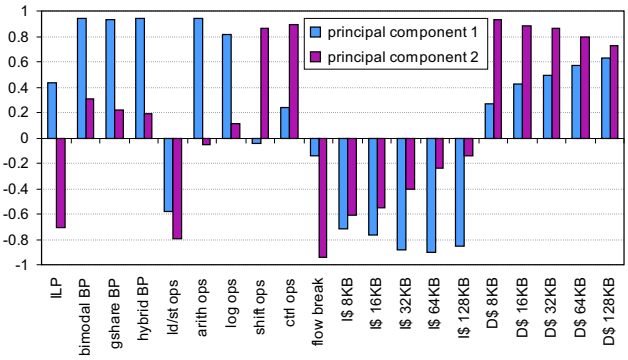

Figure 1. Factor loadings for gcc.

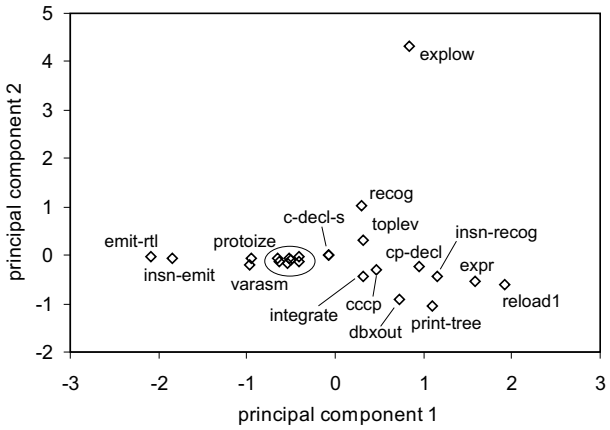

Figure 2. Workload space for gcc.

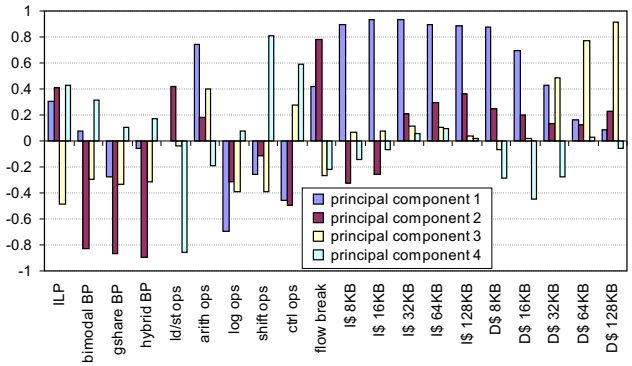

Figure 3. Factor loadings for postgres.

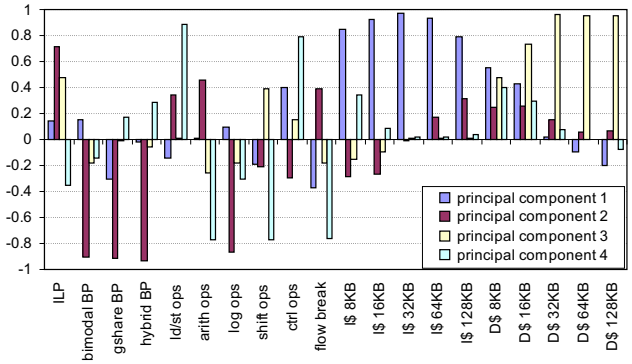

Figure 5. Factor loadings for all program-input pairs.

of the graph contains the input sets gcc, genoutput, genrecog, jump, regclass, stmt and stmt-protoize. Note that although the characteristics mentioned in Table 1 (i.e., dynamic and static instruction count, and data memory footprint) are significantly different, these input sets result in a quite similar program behavior.

4.2.2 TPC-D

PCA extracted four principal components from the data of postgres running 16 TPC-D queries, accounting for 96.2% of the total variance; The first component accounts for 38.7% of the total variance and is positively dominated, see Figure 3, by the percentage of arithmetic operations, the I-cache miss rate and the D-cache miss rate for small cache sizes; and negatively dominated by the percentage of logical operations. The second component accounts for 24.7% of the total variance and is postively dominated by the number of instructions between two taken branches and negatively dominated by the branch prediction accuracy. The third component accounts for 16.3% of the total variance and is positively dominated by the D-cache miss rates for large cache sizes. The fourth component accounts for 16.4% of the total variance and is positively dominated by the percentage of shift operations and negatively dominated by the percentage memory operations.

Figure 4 shows the data points of postgres running the TPC-D queries in the 4-dimensional space built up by these four (rotated) components. To display this 4-dimensional space understandably, we have shown the first principal component versus the second in one graph; and the third versus the fourth in another graph. This graph does not reveal a strong clustering among the various queries. From this graph, we can also conclude that some queries exhibit a significantly different behavior than the other queries. For example, queries 7 and 8 have significantly higher D-cache miss rates for large cache sizes. Query 16 has a higher percentage of shift operations and a lower percentage of load/store operations.

4.2.3 Workload Space

Now we change the scope to the entire workload space. PCA extracts four principal components from the data of

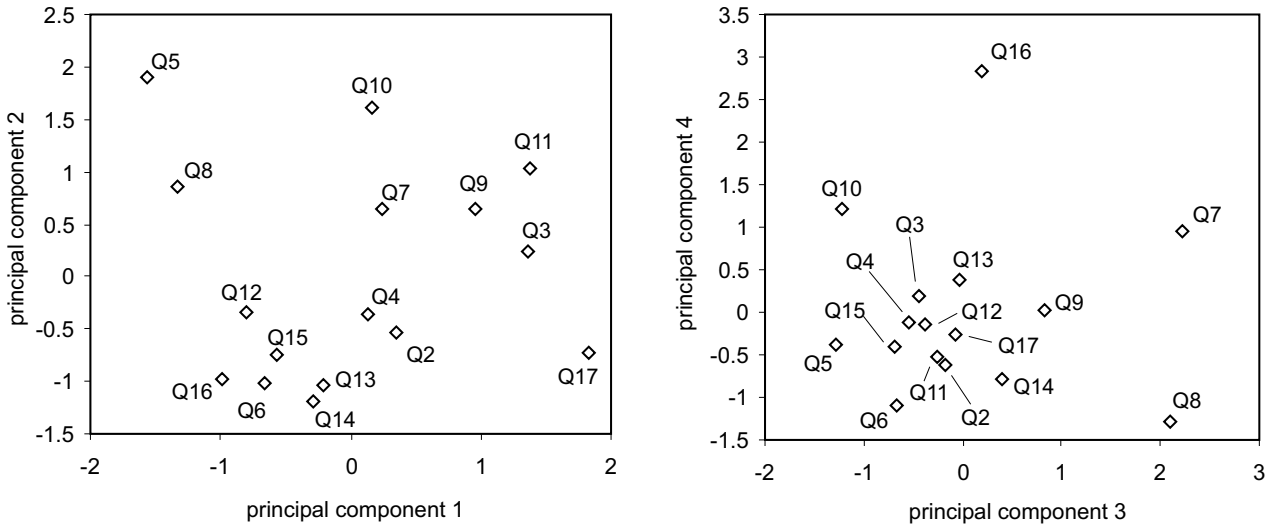

Figure 4. Workload space for postgres: first component vs. second component (graph on the left) and third vs. fourth component (graph on the right).

all 79 benchmark-input pairs as described in Table 1, accounting for 93.1% of the total variance. The first component accounts for 26.0% of the total variance and is positively dominated, see Figure 5, by the I-cache miss rate. The second principal component accounts for 24.9% of the total variance and is positively dominated by the amount of ILP and negatively dominated by the branch prediction accuracy and the percentage of logical operations. The third component accounts for 21.3% of the total variance and is positively dominated by the D-cache miss rates. The fourth component accounts for 20.9% of the total variance and is positively dominated by the percentage of load/store and control operations and negatively dominated by the percentage of arithmetic and shift operations as well as the number of instructions between two sequential flow breaks.

The results of the analyses that were done on these data, are shown in Figures 6 and 7. Figure 6 represents the program-input pairs in the 4-dimensional workload space built up by the four retained principal components. The dendrogram corresponding to the cluster analysis is shown in Figure 7. Program-input pairs connected by small linkage distances are clustered in early iterations of the analysis and thus, exhibit 'similar' behavior. Program-input pairs on the other hand, connected by large linkage distances exhibit different behavior.

Isolated points. From the data presented in Figures 6 and 7, it is clear that benchmarks go, ijpeg and compress are isolated in this 4-dimensional space. Indeed, in the dendrogram shown in Figure 7, these three benchmarks are connected to the other benchmarks through long linkage distances. E.g., go is connected to the other benchmarks with a linkage distance of 12.8 which is much larger than the linkage distance for more strongly clustered pairs, e.g., 2 or 4. An explanation for this phenomenon can be found in Figure 6. Compress discriminates itself along the third component which is due to its high D-cache miss rate. For ijpeg, the different behavior is due to, along the fourth component, the high percentage of arithmetic and shift operations, the high number of instructions between two taken branches and the low percentage of load/store and control operations. For go the discrimination is made along the second component or the low branch prediction accuracy, the low percentage of logical operations and the high amount of ILP.

Strong clusters. There are also several strong clusters which suggests that only a small number (or in some cases, only one) of the input sets should be selected to represent the whole cluster. This will ultimately reduce the total simulation time since only a few (or only one) program-input pairs need to be simulated instead of all the pairs within that cluster. We can identify several strong clusters:

- The data points corresponding to the gcc benchmark are strongly clustered, except for the input sets emit-rtl, insn-emit and explow. These three input sets exhibit a different behavior from the rest of the input sets. However, emit-rtl and insn-emit have a quite similar behavior.
- The data points corresponding to the lisp interpreter li except for browse, boyer and takr are strongly clustered as well. This can be clearly seen from Figure 7 where this group is clustered with a linkage distance that is smaller than 2. The three input sets with a different behavior are grouped with the other li input sets with a linkage distance of approximately 3. The variety within li is caused by the data cache miss rate measured by the third principal component, see Figure 6.

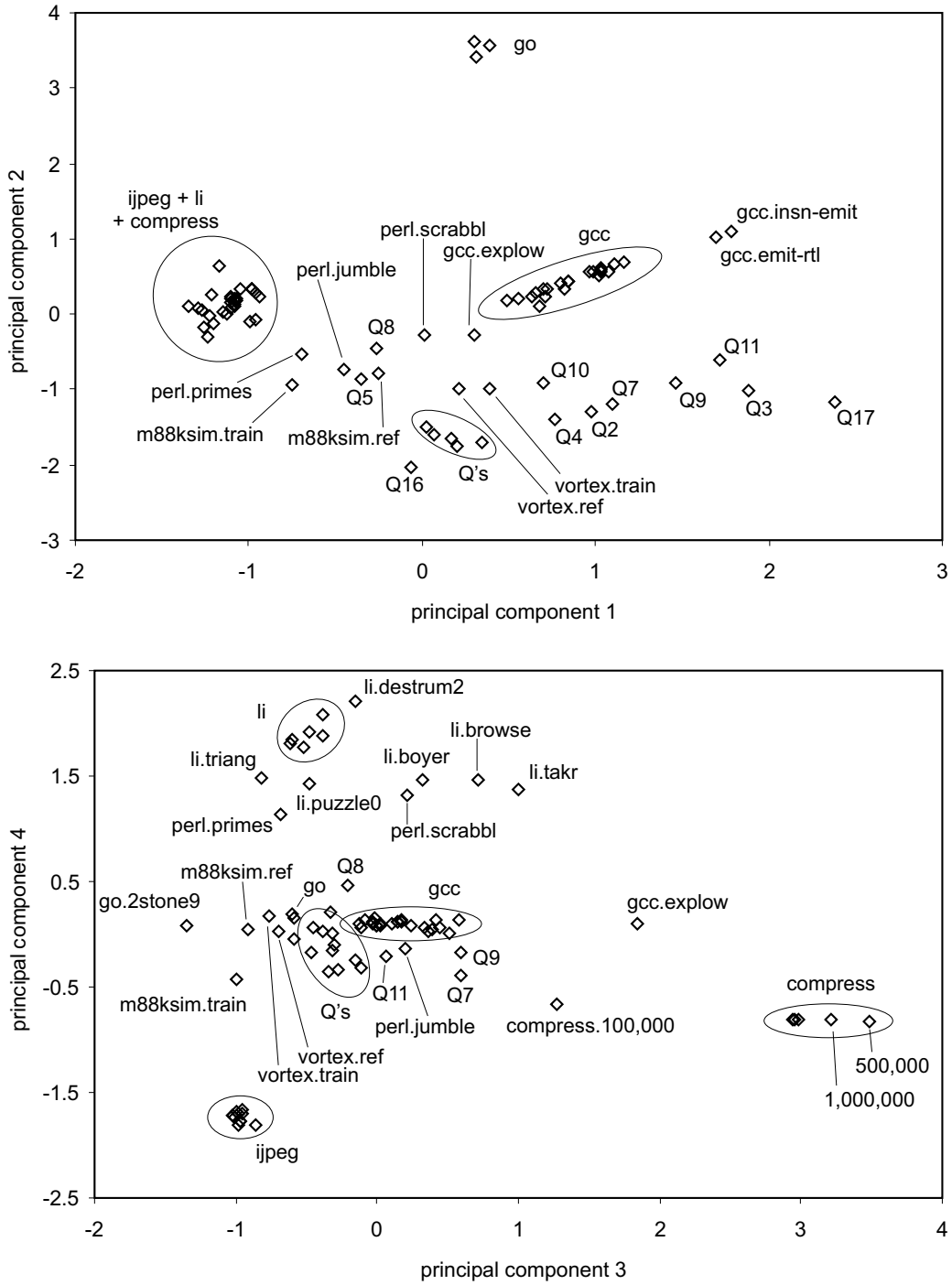

Figure 6. Workload space for all program-input pairs: first component vs. second component (upper graph) and third vs. fourth component (bottom graph).

- According to Figure 7, there is also a small cluster containing TPC-D queries, namely queries 6, 12, 13 and 15.
- All input sets for ijpeg result in similar program behavior since all input sets are clustered in one group. An important conclusion from this analysis is that in spite of the differences in image dimensions, ranging from small images (512x482) to large images (2362x1570),

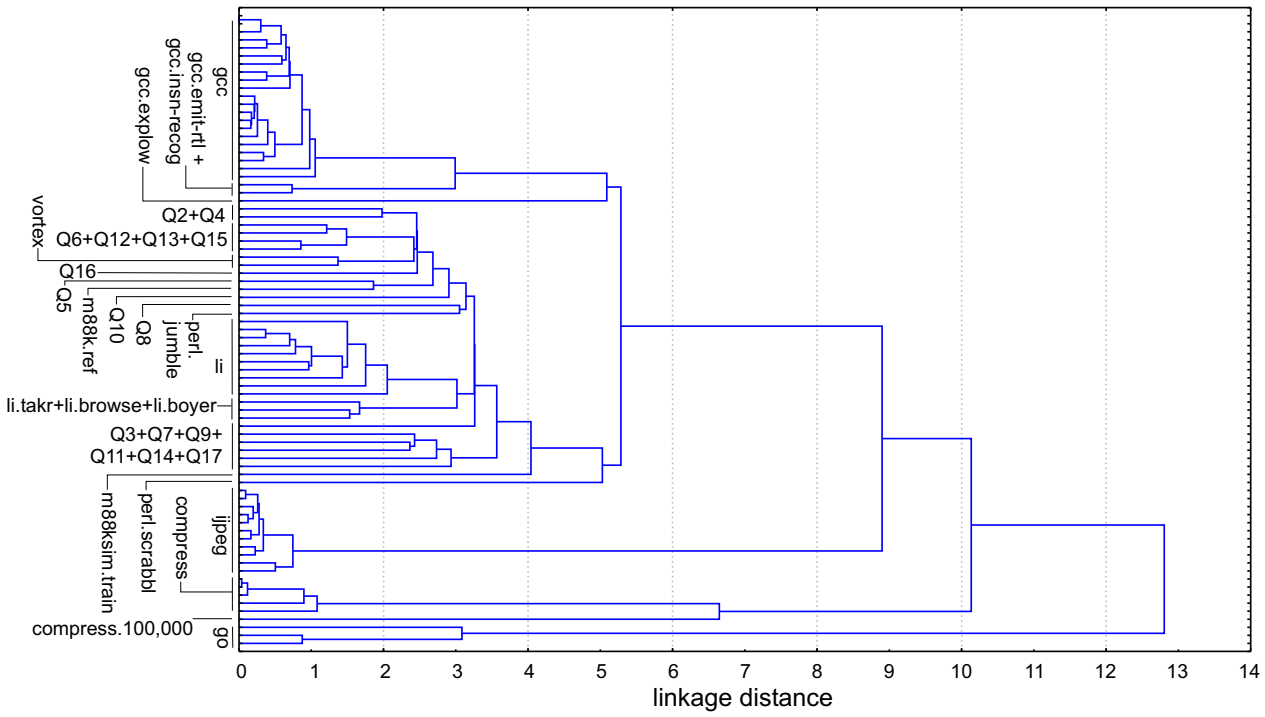

Figure 7. Cluster analysis.

the behavior of ijpeg remains quite the same.

- The input sets for compress are strongly clustered as well except for '100000 e 2231'.

Reference vs. train inputs. Along with its benchmark suite SPECint, SPEC releases reference and train inputs. The purpose for the train inputs is to provide input sets that should be used for profile-based compiler optimizations. The reference input is then used for reporting results. Within the context of this paper, the availability of reference and train input sets is important for two reasons. First, when reference and train inputs result in similar program behavior we can expect that profile-driven optimizations will be effective. Second, train inputs have a smaller dynamic instruction counts which make them candidates for more efficient simulation runs. I.e., when a train input exhibits a similar behavior as a reference input, the train input can be used instead of the reference input for exploring the design space which will lead to a more efficient design flow.

In this respect, we take the following conclusions:

- The train and reference input for vortex exhibit similar program behavior.
- For m88ksim on the other hand, this is not true.
- For go, the train input '50 9 2stone9.in' leads to a behavior that is different from the behavior of the reference inputs '50 21 9stone21.in' and '50 21 5stone21.in'. The two reference inputs on the other hand, have a quite similar behavior.
- All three inputs for perl (two reference inputs and one train input) result in quite different behavior.

Reduced inputs. KleinOsowski *et al.* [8] propose to reduce the simulation time of benchmarks by using reduced input sets. The final goal of their work is to identify a reduced input for each benchmark that results in similar behavior as the reference input but with a significant reduction in dynamic instruction counts and thus simulation time. From the data in Figures 6 and 7, we can conclude that, e.g., for ijpeg this is a viable option since small images result in quite similar behavior as large images. For compress on the other hand, we have to be careful: the reduced input '100000 e 2231' which was derived from the reference input '14000000 e 2231' results in quite different behavior. The other reduced inputs for compress lead to a behavior that is similar to the reference input.

Impact of input set on program behavior. As stated before, this analysis is useful for identifying the impact of input sets on program behavior. For example:

- The data points corresponding to postgres running the TPC-D queries are weakly clustered. The spread along the first principal component is very large and covers a large fraction of the first component. Therefore, a wide range of different I-cache behavior can be observed when running the TPC-D queries. Note also that all the queries result in an above-average branch

prediction accuracy, a high percentage of logical operations and low ILP (negative value along the second principal component).

- The difference in behavior between the input sets for compress is mainly due to the difference in the data cache miss rates (along the third principal component).
- In general, the variation between programs is larger than the variation between input sets for the same program. Thus, when composing a workload, it is more important to select different programs with a well chosen input set than to include various inputs for the same program. For example, the program-input pairs for gcc (except for explow, emit-rtl and insn-emit) and ijpeg are strongly clustered in the workload space. In some cases however, for example postgres and perl, the input set has a relatively high impact on program behavior.

4.3 Preliminary validation

As stated before, the purpose of the analysis presented in this paper is to identify clusters of program-input pairs that exhibit similar behavior. We will show that pairs that are close to each other in the workload space indeed exhibit similar behavior when changes are made to the microarchitecture on which they run.

In this section, we present a preliminary validation in which we observe the behavior of several input sets for gcc and one input set of each of the following benchmarks: go and li. The reason for doing a validation using a selected number of program-input pairs instead of all 79 program-input pairs is to limit simulation time. The simulations that are presented in this section already took several weeks. As a consequence, simulating all program-input pairs would have been impractically long[1]. However, since gcc presents a very diverse behavior (strong clustering versus isolated points, see Figure 2), we believe that a succesful validation on gcc with some additional program-input pairs can be extrapolated to the complete workload space with confidence.

We have used seven input sets for gcc, namely explow, insn-recog, gcc, genoutput, stmt, insn-emit and emit-rtl. According to the analysis done in section 4.2.1, emit-rtl and insn-emit should exhibit a similar behavior; the same should be true for gcc, genoutput and stmt. explow and insn-recog on the other hand, should result in a different program behavior since they are quite far away from the other input sets that are selected for this analysis. For go and li, we used 50 9 2stone9.in and boyer, respectively.

We used SimpleScalar v3.0 [2] for the Alpha architecture as simulation tool for this analysis. The baseline architecture has a window size of 64 instructions and an issue width of 4.

In Figures 8, 9, 10 and 11, the number of instructions retired per cycle (IPC) is shown as a function of the I-cache configuration, the D-cache configuration, the branch predictor and the window size versus issue width configuration,

[1]This is exactly the problem we are trying to solve.

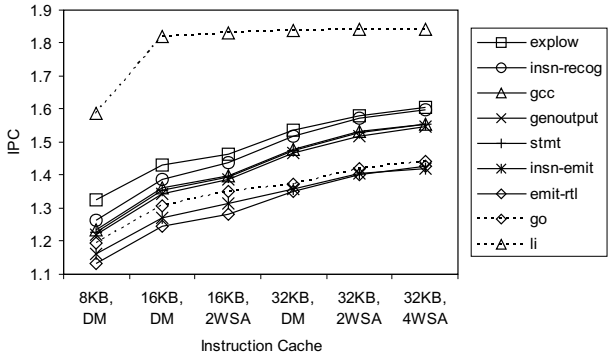

Figure 8. IPC as a function of the I-cache configuration; 16KB DM D-cache and 8K-entry bimodal branch predictor.

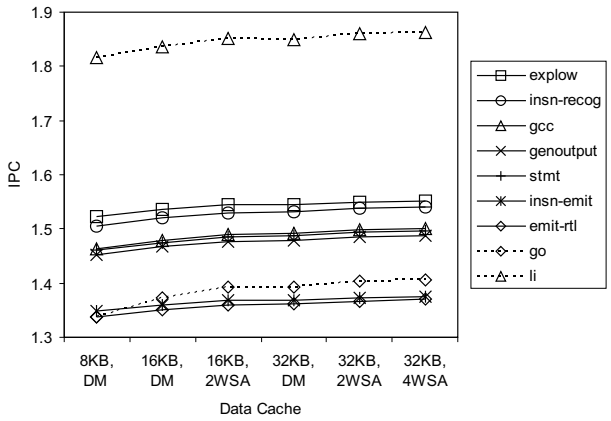

Figure 9. IPC as a function of the D-cache configuration; 32KB DM I-cache and 8K-entry bimodal branch predictor.

respectively. We will first discuss the results for gcc. Afterwards, we will detail on the other benchmarks.

For gcc, we clearly identify three groups of input sets that have similar behavior, namely (i) explow and insn-recog, (ii) gcc, genoutput and stmt, and (iii) insn-emit and emit-rtl. For example, in Figure 10, the branch behavior of group (i) is significantly different from the other input sets. Or, in Figure 11, the scaling behavior as a function of window size and issue width is quite different for all three groups. This could be expected for groups (ii) and (iii) as discussed earlier. The fact that explow and insn-recog exhibit similar behavior on the other hand, is unexpected since these two input sets are quite far away from each other in the workload space, see Figure 2. The discrimination between these two input sets is primarily along the second compo-

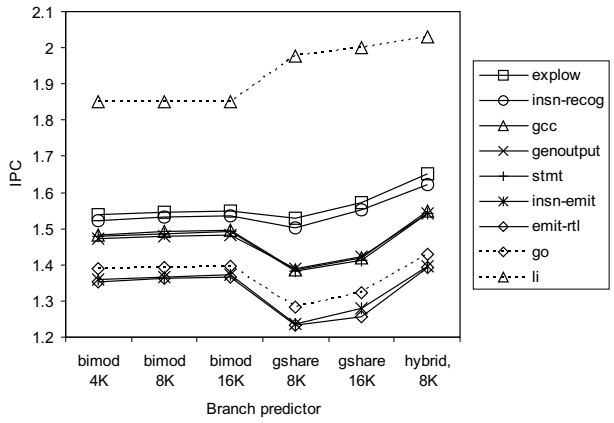

Figure 10. IPC a function of the branch predictor configuration; 32KB DM I-cache and 32KB DM D-cache.

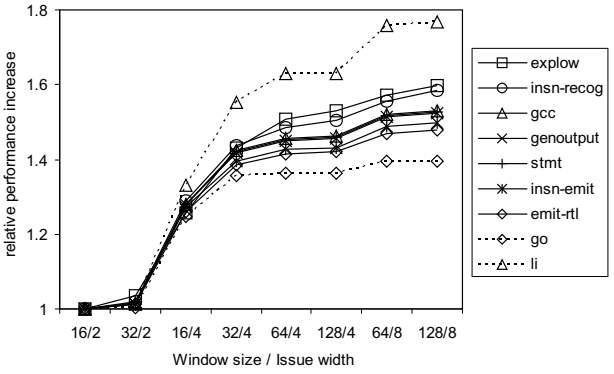

Figure 11. Performance increase w.r.t. the 16/2 configuration as a function of the window size and the issue width; 32KB DM I-cache, 32KB DM D-cache and 8K-entry branch predictor.

nent. Along the first component on the other hand, explow and insn-recog have a similar value. This leads us to the conclusion that the impact on performance of the program characteristics measured along the second principal component is smaller than along the first component.

The other two benchmarks, go and li, clearly exhibit a different behavior on all four graphs. This could be expected from the analysis done in section 4.2.3 since PCA and cluster analysis pointed out that these benchmarks have a different behavior. Most of the mutual differences can be explained from the analysis done in this paper. For example, go has a different D-cache behavior than gcc which is clearly reflected in Figure 9. Also, li has a different I-cache behavior than gcc and go which is reflected in Figure 8. Other differences however, are more difficult to explain. Again, this phenomenon is due to the fact that some microarchitectural parameters have a minor impact on performance for a given microarchitectural configuration. However, for other microarchitectural configurations we can still expect different behavior. For example, go has a different branch behavior than gcc, according to the analysis done in section 4.2.3; in Figure 10, go and gcc exhibit the same behavior.

5 Related work

Saavedra and Smith [11] addressed the problem of measuring benchmark similarity. For this purpose they presented a metric that is based on dynamic program characteristics for the Fortran language, for example the instruction mix, the number of function calls, the number of address computations, etc. For measuring the difference between benchmarks they used the squared Euclidean distance. The methodology in this paper differs from the one presented by Saavedra and Smith [11] for two reasons. First, the program characteristics measured here are more suited for performance prediction of contemporary architectures since we include branch prediction accuracy, cache miss rates, ILP, etc. Second, we prefer to work with uncorrelated program characteristics (obtained after PCA) for quantifying differences between program-input pairs, as extensively argued in section 3.3.

Hsu *et al.* [5] studied the impact of input data sets on program behavior using high-level metrics, such as procedure level profiles and IPC, as well as low-level metrics, such as the execution paths leading to data cache misses.

KleinOsowski *et al.*[8] propose to reduce the simulation time of the SPEC 2000 benchmark suite by using reduced input data sets. Instead of using the reference input data sets provided by SPEC, which result in unreasonably long simulation times, they propose to use smaller input data sets that accurately reflect the behavior of the full reference input sets. For determining whether two input sets result in more or less the same behavior, they used the chi-squared statistic based on the function-level execution profiles for each input set. Note that a resemblance of function-level execution profiles does not necessarily imply a resemblance of other program characteristics which are probably more directly related to performance, such as instruction mix, cache behavior, etc. The latter approach was taken in this paper for exactly that reason. KleinOsowski *et al.* also recognized that this is a potential problem. The methodology presented in this paper can be used as well for selecting reduced input data sets. A reference input set and a resembling reduced input set will be situated close to each other in the q-dimensional space built up by the principal components.

Another possible application of using a data reduction technique such as principal components analysis, is to compare different workloads. In [3], Chow *et al.* used PCA to compare the branch behavior of Java and non-Java workloads. The interesting aspect of using PCA in this context is that PCA is able to identify on which point two workloads differ.

Huang and Shen [6] evaluated the impact of input data sets on the bandwidth requirements of computer programs.

Changes in program behavior due to different input data sets are also important for profile-guided compilation [12], where profiling information from a past run is used by the compiler to guide its optimisations. Fisher and Freudenberger [4] studied whether branch directions from previous runs of a program (using different input sets) are good predictors of the branch directions in future runs. Their study concludes that branches generally take the same directions in different runs of a program. However, they warn that some runs of a program exercise entirely different parts of the program. Hence, these runs cannot be used to make predictions about each other. By using the average branch direction over a number of runs, this problem can be avoided. Wall [15] studied several types of profiles such as basic block counts and the number of references to global variables. He measured the usefulness of a profile as the speedup obtained when that profile is used in a profile-guided compiler optimisation. Seemingly, the best results are obtained when the same input is used for profiling and measuring the speedup. This implies that every input is different in some sense and leads to different compiler optimisations.

6 Conclusion

In microprocessor design, it is important to have a representative workload to make correct design decisions. This paper proposes the use of principal components analysis and cluster analysis to efficiently explore the workload space. In this workload space, benchmark-input pairs can be displayed and a distance can be computed that gives us an idea of the behavioral difference between these benchmark-input pairs. This representation can be used to measure the impact of input data sets on program behavior. In addition, our methodology was succesfully validated by showing that program-input pairs that are close to each other in the principal components space, indeed exhibit similar behavior as a function of microarchitectural changes. Interesting applications for this technique are the composition of workloads and profile-based compiler optimizations.

Acknowledgements

Lieven Eeckhout and Hans Vandierendonck are supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT). The authors would also like to thank the anonymous reviewers for their valuable comments.

References

[1] P. Bose and T. M. Conte. Performance analysis and its impact on design. *IEEE Computer*, 31(5):41–49, May 1998.

[2] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set. Computer Architecture News, 1997. Version 2.0. See also `http://www.simplescalar.org` for more information.

[3] K. Chow, A. Wright, and K. Lai. Characterization of Java workloads by principal components analysis and indirect branches. In *Proceedings of the Workshop on Workload Characterization (WWC-1998), held in conjunction with the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pages 11–19, Nov. 1998.

[4] J. Fisher and S. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 85–95, 1992.

[5] W. C. Hsu, H. Chen, P. Y. Yew, and D.-Y. Chen. On the predictability of program behavior using different input data sets. In *Proceedings of the Sixth Workshop on Interaction between Compilers and Computer Architectures (INTERACT 2002), held in conjunction with the Eighth International Symposium on High-Performance Computer Architecture (HPCA-8)*, pages 45–53, Feb. 2002.

[6] A. S. Huang and J. P. Shen. The intrinsic bandwidth requirements of ordinary programs. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 105–114, Oct. 1996.

[7] L. K. John, P. Vasudevan, and J. Sabarinathan. Workload characterization: Motivation, goals and methodology. In L. K. John and A. M. G. Maynard, editors, *Workload Characterization: Methodology and Case Studies*. IEEE Computer Society, 1999.

[8] A. J. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Workload Characterization of Emerging Computer Applications, Proceedings of the IEEE 3rd Annual Workshop on Workload Characterization (WWC-2000) held in conjunction with the International Conference on Computer Design (ICCD-2000)*, pages 83–100, Sept. 2000.

[9] B. F. J. Manly. *Multivariate Statistical Methods: A primer*. Chapman & Hall, second edition, 1994.

[10] S. McFarling. Combining branch predictors. Technical Report WRL TN-36, Digital Western Research Laboratory, June 1993.

[11] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, Nov. 1996.

[12] M. D. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–11, 2000.

[13] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Western Research Lab, Compaq, Mar. 1994.

[14] I. StatSoft. STATISTICA for Windows. Computer program manual. 1999. `http://www.statsoft.com`.

[15] D. W. Wall. Predicting program behavior using real or estimated profi les. In *Proceedings of the 1991 International Conference on Programming Language Design and Implementation (PLDI-1991)*, pages 59–70, 1991.